



Overview

For this semester, we are going to organize a series of video games. This will allow you to run gaming tournaments, sell retro games, and much more.

In your first project, you created a base Game Class. Now, it is time to create your first data structure to maintain a collection. In particular, you will create a well-written, efficient, and versatile Linked-List class. You will use this in a future assignment to create an equally efficient Stack, Queue, and much more.

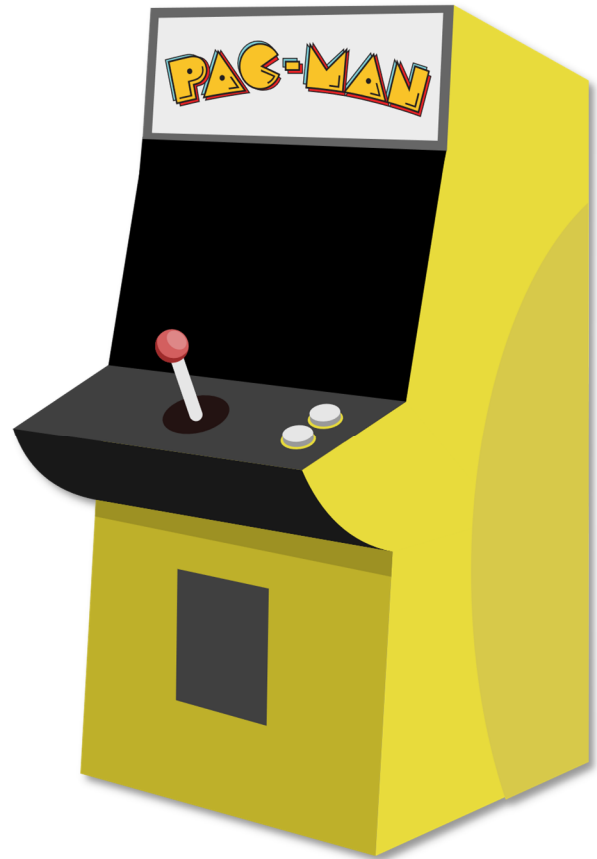
Given you already have developed excellent programming skills in CSc 20, this shouldn't be a difficult assignment.

Part 1: Node Class

You are going to create a very, very basic node class. Make sure to come up with a nice, well-documented, and well-written solution. We may build upon it, later, to create more advanced data structures.

This class must be encapsulated (i.e. hidden) within the GameList Class (see below). **Make it private.**

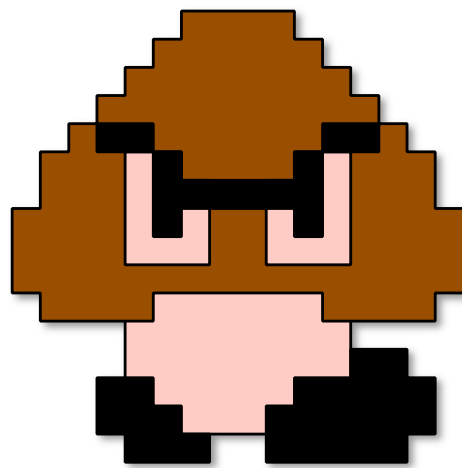
Since this class will be hidden inside the Linked List class, you can make these fields "public" and not worry about assessors and mutators (get and set).



Node Class Attributes		
Node	next	The next node in the chain.
Game	value	The contents of the node

The following are the required constructors. You can create more if you like. Practically all implementations of nodes contain these two.

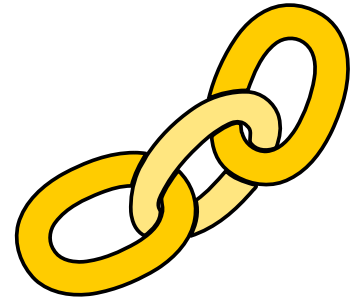
Node Class Constructors	
Node (Game)	Next is set to null
Node (Game, Node)	Next is set to the next in the series.



A fungi and a fun guy!

Part 2: Game List Class

You are going to write a singly linked-list class. You will use this again in the future. So, do a good job on it, or you will hurt your future assignments. The GameList Class should maintain two private fields to maintain the list. In particular, it needs to maintain the link to the head and tail.



GameList Private Fields	
Node	head
Node	tail

Your methods **must** be $O(1)$ for all adds. So, you **must** maintain a reference (link) to the tail. The following is the required public methods (aka the classes interface – all classes define an interface).

GameList Class Public Methods		
String	about ()	Returns text about you – the author of this class.
void	addHead (Game)	Adds the node to the head of the list. This must be $O(1)$.
void	addTail (Game)	Adds the node to the tail of the list. This must be $O(1)$.
boolean	isEmpty ()	Returns true if the linked list is empty. In other words, the head is null.
Game	removeHead ()	Remove the head from the linked list.
Game	find (int)	Finds and returns a game with the specified ID. If no instance is found, return null.
GameList	sublist (Category)	Returns a new instance of the GameList class containing all the objects of the specified Category. The current list must not be altered.
GameList	sublist (Platform)	Returns a new instance of the GameList class containing all the objects of the specified Platform. The current list must not be altered.
String	toString ()	Returns the contents of the linked list, from first to last, with each item separated by a new line. Call the description() method on each item.

Part 3: Testing

1. You need to test the logic of your GameList class. In the **main()** function of your Tester Class (or whatever you call it – it doesn't matter to me)
2. Create at least **fifteen (15)** instances of the Game Class.
3. Add them to, at least, **two** instances of the GameList Class. Use make use of the add and remove methods. Keep using the **toString()** method to see if you get the correct results.
4. Also test the **sublist()** methods and make sure that they don't destroy the data in the main list. Do **not** read values from a file or the keyboard.

Hints

- Write, and test, the add and remove functions first. These are the most essential, and, if they don't work, the rest of the methods won't work.
- The **toString()** method is absolutely vital to testing. You can use this method when debugging the add and remove functions. It can also be used to test the sublist methods.
- All the methods specified above will come up later.
- The two **sublist** methods and the **find** method will use while loops. Please see the lecture notes for the necessary logic.



Allowed Programming Languages

You may use any of the following programming languages.

- C#
- Java

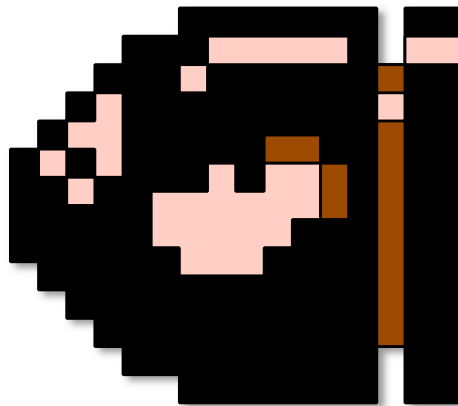
The following **cannot** be used:

- | | | |
|--------------|----------|---------------------|
| • C++ | • Lua | • Scala |
| • Groovy | • Nim | • Swift |
| • JavaScript | • Python | • TypeScript |
| • Kotlin | • Ruby | • Visual Basic .NET |

Requirements

The following are the requirements for this assignment:

- This **must** be **completely** all your code. If you share your solution with another student or re-use code from another class, you will receive a zero.
- **Do not** use any built-in collection classes such as ArrayList. Many programming languages have nice collection classes available. Do not use any of them. **If you use any of these, you will receive a zero. No exceptions. No resubmissions.**
- You **must** write your own Linked-List. It must all be **O(1)** for all adds.
- You may use any of the programming languages listed above.
- Create some excellent testing for your class.
- Proper style (see below).
- Do **not** put your **main()** method inside the GameList class. This will cause major issues going forward.



An enemy of the highest caliber!

Due Date & Submission

Due **October 9, 2026** by **11:59 pm**.

E-Mail your source code files to dcook@csus.edu. Please zip your source code files into a single .zip or .tar file. Please take note of the following. If you send your files incorrectly, I either won't receive them or won't be able to grade them.



1. Send **ONLY** .java, or .cs files. The e-mail server will delete all attachments that have file extensions it deems dangerous. This includes .jar, .exe, and .class.
2. Do **NOT** simply zip an entire folder. The server will delete it.
3. Do **NOT** send a cloud link. I cannot open or grade these. Send an attachment.
4. Do **NOT** send it to canvas. I will not read nor grade Canvas e-mails.

Grading

1	Node class is encapsulated in the GameList class (private)	10%
2	Correct add methods – must be O(1)	20%
3	Correct sublist methods	20%
4	Correct find method	20%
5	Proper Style	10%
6	Proper testing (see above).	10%
7	<code>main()</code> is in a testing class (not in Game, Node or GameList)	10%



Proper Style

Points will be deducted if your program doesn't adhere to basic programming style guidelines.

1. If programming C++, Java, or C#, I don't care where you put the starting curly bracket. Just be consistent.
2. Indentation must be used.
3. Indentation must be consistent. Three or four spaces works. **Beware of the tab character!** It might not appear correctly on my computer (tabs are inconsistent in size).
4. Proper commenting. Not every line needs a comment, but sections that contain logic often do. Add a comment before every section of code – such as a loop or If Statement. Any complex idea, such as setting a link, must have a comment.

Beware of Static

The **static** keyword in Java indicates something that will have just 1 copy and it is not associated with an instance.

- If you define a static class, then you cannot create instances of it. Java will also force you to make every function static as well.
- A static function is associated with the class, but not an instance. Normally, your program has a static function called `main()`. This should be the **only** static definition in your program.
- If you declare fields in a class as static, there is only one copy that is shared by all instance. This can be disastrous in your code. For example, if you declare `head` and `tail` as static, you probably won't get weird side effects if you create **only one** instance of your class. However, if you create more than one instance, **they will fail** (they are using each other's data).



Some Helpful Pseudocode

AddHead Pseudocode

```
procedure AddHead(Game game)
    node = new Node(game)          ... Create a new node.

    if the list is empty           ... is head null?
        this.head = node           ... Link both head and tail to the new node
        this.tail = node
    else
        node.next = this.head      ... Link the new node to the head
        this.head = node           ... The head is now the new node
    end if
end procedure
```

AddTail Pseudocode

```
procedure AddTail(Game game)
    node = new Node(game)          ... Create a new node.

    if the list is empty           ... is head/tail null?
        this.head = node
        this.tail = node
    else
        this.tail.next = node      ... Link the old tail to the new node
        this.tail = node           ... The new node is the tail
    end if
end procedure
```

RemoveHead Pseudocode

function RemoveHead() returns Game

```

    if the list is empty                ... is head null?
        result = null                  ... We could also throw an error
    else if (this.head == this.tail)    ... Just 1 node. Set head/tail to null
        result = this.head.value
        this.head = null               ... Deference both
        this.tail = null
    else                                ... 2 or more nodes.
        result = this.head.value
        this.head = this.head.next;    ... Move head to the next node
    end if

    return result
end function

```